Paper 209-2013

Working with a Large Pharmacy Database: Hash and Conquer.

David Izrael, Abt Associates Inc .

ABSTRACT

Working with a large pharmacy database means having to process - merge, sort, and summarize - hundreds of millions of observations. By themselves, traditional methods of processing can lead to prohibitive data processing times that endanger deadlines. The hash object is the fastest and most versatile method in the SAS® system of substantially accelerating the processing. In our paper, we apply hash methods to a routine lookup function where one needs to merge the kernel pharmacy database with its satellites. We also present comparatively new non-traditional features of the hash object, such as handling duplicate keys and finding frequency counters. At the same time, we underscore the necessity of traditional sort-and-merge methods, but suggest that they be used carefully.

WHAT IS THE GOAL?

The database we are dealing with consists of three large prescription data sets, one for each group of medications. Let's call them Prescriptions_A, Prescriptions_B, and Prescriptions_C. They contain the following information: RX Type, Payment Type, RX Date, Payer ID, Product-Pack Number, Product Name, RX Quantity, Days' Supply, RX Dose, Prescriber ID, Pharmacy ID, Patient ID, and others. Each of the three data sets contains around 100 million prescriptions and occupies roughly 25 GB of disk space.

Each of the three data sets is accompanied by four satellite data sets: a pharmacy reference containing characteristics like channel, Metropolitan Statistical Area (MSA), pharmacy state, pharmacy zip3, and others; a medication reference containing form, strength, and other medication characteristics; a provider reference containing the provider's specialty, zip code, and other characteristics; and a schedule reference containing the main category of the product and its schedule.

One starts, of course, by combining the master prescription data sets with their satellites. The routine practice of sorting and merging will not work in this situation. Those manipulations, when performed over such large master data sets, lead to unacceptable processing times and/or system crashes, especially in multi-user environments. Instead, we apply the method recognized by the SAS user community as most optimal when working with large databases: hash tables. For detailed descriptions of this method we direct the reader to [1], [2], [3] and to other articles which can be found online or on the SAS support site.

So, to join a master file (for instance, Prescriptions_A) with its lookup tables, we use the following code:

```
data analyt;
    set this.prescriptions_A;     /*** MASTER PRESCRIPTION DATA SETS ***/

    length form      $ 12         /*** ALLOCATE VARIABLES FROM LOOKUP TABLES ***/
           product   $ 18
           strength  $ 12
           usc       $ 5
           usc_desc  $ 30
           specialty $ 3
           zip       $ 5
           specdesc  $ 34
           specnpa   $ 4
           low_chanl   $ 1
           high_chanl  $ 1
           msa          $ 45
           pharm_state $ 2
           pharm_zip3  $ 3
           main_cat    $ 30
           schedule    $ 4 ;

if _n_ =1 then do;
   declare hash outl( dataset : 'this.pharmacy_ref');  /*** HASH TABLE: PHARMACY
REFERENCE

***/
           outl.definekey ('pharmacy_id');
           outl.definedata ('low_chanl', 'high_chanl', 'msa', 'pharm_state',
                           'pharm_zip3' );
           outl.definedone ( );

   declare hash cmf( dataset : 'this.product_ref'); /*** HASH TABLE: PRODUCT
REFERENCE***/
           cmf.definekey ('cmf10');
           cmf.definedata ('form','product','strength', 'usc', 'usc_desc');
           cmf.definedone ( );

   declare hash rxer( dataset : 'this.docs_ref');    /*** HASH TABLE: PROVIDER
REFERENCE***/
           rxer.definekey ('rxerid');
           rxer.definedata ('specialty','zip','specdesc', 'specnpa');
           rxer.definedone ( );


   declare hash categ( dataset : 'this.schedule’); /*** HASH TABLE: SCHEDULE REFERENCE
***/
           categ.definekey ('product');
           categ.definedata ('main_cat','schedule');
           categ.definedone ( );



call missing(form,
product,
strength,
usc      ,
usc_desc ,
specialty,
zip      ,
specdesc ,
specnpa,
low_chanl,
high_chanl,
msa,
pharm_state,
pharm_zip3,
main_cat,
schedule

);                  /*** INITIALIZE VARIABLES IN CASE KEYS NOT FOUND ***/
         end;
```

```
              drop rc;

         rc1 = outl.find ( );
         rc2 = cmf.find ( );
         rc3 = rxer.find ( );
         rc4 = categ.find ( );

/*** IF RESPECIVE KEY FOUND OUTPUT OBSERVATION POPULATED WITH VARIABLES SPECIFIED IN
       .definedata METHOD ***/
run;
```

All hash object syntax and methods used here should be clear for experienced readers. For inexperienced ones, there is a myriad of materials on this topic.

By using hash tables, we avoid the bottleneck effect of sorting the large data sets. As we did with Prescriptions_A, we merge Prescriptions_B and Prescriptions_C with their respective satellites. In our case, all merges are left joins because the satellite tables only have the keys that are present in the master prescription data set.

After the merge, the sizes of the three data sets climb sharply to around 70 GB of disk space. The rule of thumb in processing those data sets remains the same: avoid sorting whenever possible, leave in only necessary variables, and use hash tables.

DIGGING IN DEEPER…

Whether we compute a consumption rate for geographic variability, prevalence of providers prescribing a certain medication, or distribution of the periods between prescriptions, we are dealing with summarization. While PROC SUMMARY works just fine when there are multiple CPUs in your possession, hash methods are noticeably faster with one CPU. The following example demonstrates summation of the variable weight (each script is weighted in our database) by a pharmacy's state, county, and main category of medication (product):

```
             data hash_suminc_sum(keep = pharm_state county main_cat sum_weights     );
                  if 0 then set yourbigdataset;

                  dcl hash h(suminc: 'weight', hashexp:20, ordered: 'a');
                  h.defineKey('pharm_state', 'county', ‘main_cat’ );
                  h.defineDone();

                    do while (^ eof);
                    set yourbigdataset end = eof;
                    h.ref();
                    end;

                dcl hiter hi ('h');
                rc = hi.first();

                    do while (rc=0);
                      h.sum (sum: sum_weights);
                      output;
                      rc = hi.next ();
                    end;
                    stop;
                    run;
```

To sum values of “weight” we use the operator *suminc: ‘weight’* during the creation of the hash object. The *ref* method, new in version 9.2, is used to check for already-existing keys with certain values for *pharm_state*, *county*, and *main_cat*. If there are no keys with such values, they are added to the hash table. Using *hiter* object (hi) we set the pointer to the first record *(hi.first()*) of the hash table to retrieve hash table keys in an ascending order. Summation itself is performed by the *sum* method and stores the accumulated sums of weights to the variable *sum_weight.* The operator *output* writes the sum for certain values of *pharm_state*, *county*, and *main_cat* to the output data set *hash_suminc_sum* after which the *hi.next* operator moves to the next hash table record until it reaches the end. Finally, the operator *ordered: ‘a’* enables hash *h* to be sorted in the ascending order of *pharm_state*, *county*, and *main_cat*.

The reader should not develop the impression that the hash method is a panacea against the tribulations inherent to processing very large databases. Users will still need sorted data sets and the attributes *first.* and *last.* for tasks that are not as trivial as summation or left join. We will show how to use traditional methods in combination with hash tables. Some analyses of the pharmacy database require the data sets to be sorted by the main category of a medication, unique patient ID, and then by the script date. Older servers may allow us to do that with all the variables in the data set only when there are no other intensive I/O jobs on the server.

EVEN DEEPER…

Let us suppose that for every unique patient ID from the Prescriptions_A data set we need to retrieve all the observations for which the prescription dates for a given Patient ID are the same as the prescription dates in the Prescriptions_B data set.

First, we need to retrieve Patient _ID and all RX dates for a given medication from Prescriptions_B. This could look traditional:

```
proc sort data = Prescriptions_B(keep = patient_ID  rxdate  main_cat where =
(main_cat= 'certaindrug')) out=B (drop = main_cat rename = (rxdate = B_date));
by patient_ID rxdate;
run;
```

Since we sort a small subset of the large data set, this sort does not present any problems.

Now let us see how we could use the Prescriptions_A data set and the hash table created from the Prescriptions_B data set above to make our final retrieval.

```
data result;
drop rc;
  if _n_  = 1 then do;
dcl hash B_hash(dataset: 'work.B', multidata: ‘yes’);
B_hash.defineKey ('patient_id');
B_hash.defineData(‘patient_id’ 'B_date’);
B_hash.defineDone();
end;

set Prescriptions_A (keep = patent_id rxdate);

rc = B_hash.find();

   do while (rc=0);
     if   B_date= rxdate then output;
   rc  = B_hash.find_next();
end;
run;
```

Since the Prescriptions_B data set contains duplicate Patient IDs, the hash table *B_hash* is created with the option *multidata: 'yes'* which allows the hash table to store the duplicate keys. After the hash table is loaded, the first *find* operator locates *patient_id* in the hash table from the Prescriptions_A data set.

In a DO WHILE loop, *B_date* of every following record in the hash table is compared with *rxdate* of found *patient_id* and if it is the same as *B_date* the output observation is created. The *find_next* method finds the next record in the hash table with the same Patient ID, and dates are compared again until a new Patient ID is encountered in the hash table.

However, if we are know that the subset of the Prescriptions_B data set consisting of *patient_id* and *rxdate* can be incorporated into a hash table, the same goal could be achieved much faster. Let us first estimate how much memory such a hash table would take up. *patient_id* (8 digits) and *rxdate* (5 digits) might take up 4+4= 8 bytes. Thus, for 100 million pairs of Patient ID - RX Date, the hash table requires less than 1 GB, which is quite realistic nowadays. Our code above would turn into the following:

```
data result;
    set this.prescriptions_A;    /*** MASTER PRESCRIPTION DATA SETS ***/

if _n_ =1 then do;
   declare hash large_to_small( dataset : 'this.Prescriptions_B', hashexp: 20);  /***
hash
                                                              table: pharmacy
reference ***/
           large_to_small.definekey ('Patient_ID', 'RXdate');
           large_to_small.definedata ('Patient_ID', 'RXdate');
           large_to_small.definedone ( );
         end;

               drop rc;

          rc = large_to_small.find ( );
run;
```

By using the operator *hashexp: 20* we request the maximum amount of memory for our hash table. Also, there is no need to specify *multidata: 'yes'* since the Patient ID - RX Date pair is a unique composite key.

YET, DEEPER…

In the following example we will demonstrate how to use the Prescriptions_A data set and the two hash tables loaded from the Prescriptions_B and Prescriptions_C data sets to output *patient_id* and *rxdate* of the Prescriptions_A data set (those that overlap with RX dates from the Prescriptions_B and Prescriptions_C data sets for a given Patient ID). In other words, we are interested in the Prescriptions_A patients who also take the other two types of medications within the same time span.

```
data patient_id_rxdate_mixture;
drop rc;
  if _n_  = 1 then do;

dcl hash B_hash(dataset: 'Prescriptions_B', multidata: 'yes'); /* LOAD HASH TABLE FROM
                                                                 PRESCRIPTION_B
                                                                 DATABASE */
```

```
B_hash.defineKey ('patient_id');
B_hash.defineData('patient_id' 'B_date');
B_hash.defineDone();

dcl hash C_hash(dataset: 'Prescriptions_C', multidata: 'yes'); /* LOAD HASH TABLE FROM
                                                                  PRESCRIPTION_C
DATABASE */
C_hash.defineKey ('patient_id');
C_hash.defineData('patient_id' 'C_date');

C_hash.defineDone();

end;

set Prescriptions_A (keep = patent_id rxdate);

Earlier_than_B=.; Later_than_B=.; Earlier_than_C=.; Later_than_C=.;  /* INITIALIZE
FLAGS FOR
                                                                        DATE
                                                                        COMPARISON */

rc_B = b_hash.find ();                  /* FIND PATIENT ID IN B_HASH TABLE */

do while (rc_B=0);
    if rxdate >= B_date then Later_than_B=1;    /** COMPARE RXDATE WITH B DATES IN HASH
B **/
    if rxdate <= B_date then Earlier_than_B=1;
    rc_B = B_hash.find_next();                  /** MOVE TO NEXT DATE IN HASH B **/
end;

rc_C = c_hash.find ();                  /* FIND PATIENT ID IN C_HASH TABLE */

do while (rc_C=0);
    if rxdate >= C_date then Later_than_C=1;    /** COMPARE RXDATE WITH C DATES IN HASH
C **/
    if rxdate <= C_date then Earlier_than_C=1;
    rc_C = C_hash.find_next();                  /** MOVE TO NEXT DATE IN HASH C **/
end;

/* OUTPUT IF CURRENT RXDATE WITHIN TIME SPAN OF B AND C DATES IN HASHES B AND C */

if later_than_B=1 and earlier_than_B=1 and later_than_C=1 and earlier_than_C=1 then
output;
run;
```

SAS provides a vast number of ways of processing data from very large databases. Secosky and Bloom (2007) present a comprehensive table of pros and cons for the most popular methods. By following this table, users will gain immensely from analyzing their possibilities for each particular case.

REFERENCES

[1] Secosky, Jason, and Janice Bloom. 2007. "Getting Started with the DATA Step Hash Object." *Proceedings of the First SAS Global Forum.* Cary, NC: SAS Institute Inc. Available at www2.sas.com/proceedings/forum2007/271-2007.pdf

[2] Dorfman, Paul and Vyverman, Koen. 2005. “Data Step Hash Objects as Programming Tools.” *Proceedings of the Thirtieth Annual SAS Users Group International Conference*. Available

http://www2.sas.com/proceedings/sugi30/236-30.pdf

[3] Ray, Robert and Jason Secosky (2008). "*Better Hashing in SAS® 9.2," Proceedings of the Second Annual SAS Global Forum (SGF) Conference, SAS Institute Inc., Cary, NC, USA.*
support.sas.com/resources/papers/sgf2008/hashing92.pdf

DISCLAIMER

CONTACT INFORMATION

David Izrael,
Abt Associates,
55 Wheeler Street,
Cambridge, MA 02138
Ph. (O): 617.349.2434
E-mail: david_izrael@abtassoc.com